Managing Complexity in Scientific Programs in Fortran95

Viktor K. Decyk, UCLA

Abstract

The complexity of scientific software is increasing as computer power is rising. Such complexity becomes unmanageable without the help of more sophisticated programming language features. This talk will discuss some of the features of Fortran95 which are useful for managing complexity, largely by using ideas from object-oriented programming. Examples will be given from the UCLA Parallel PIC Framework (UPIC) and the Summit Framework.

Outline:

Some new features in Fortran95:

- Modules, Array Objects, Derived Types

Object-Oriented Concepts in Fortran95

- Information Hiding and Data Encapsulation
- Abstract Datatypes, Classes and Objects
- Inheritance
- Dynamic Dispatch or Run-Time Polymorphism

UPIC Framework

Modules

A module is a new program unit for grouping together subroutines, data, etc. We will use this construct to build classes later.

```
module my_module
 real, parameter :: pi = 3.14159      ! data goes here
  contains                            ! procedures go here
  subroutine roots(a,b,state)
 implicit none
 real, intent(out) :: a
 real, intent(in) :: b
 integer, intent(inout) :: state
 ...          ! contents of procedure go here
            ! Something new: pi is available here
 end subroutine roots
end module my_module

program my_program
use my_module  ! make information in module available
               ! pi and roots are now known here
 real :: x = 1.0, y = 2.0, z = 3.0
 integer :: s = 1
                      ! the arguments will be checked
call roots(x,y,z)     ! interface block no longer needed
                      ! z is still the wrong type.
end program
```

Modules can be compiled separately. Modules are extremely useful for managing complexity.

Array Objects

Arrays in Fortran90 are actually self-contained objects. They contain hidden information about their sizes and shapes. As a result of this extra information, one does not have to explicitly declare array dimensions in subroutines.

```
subroutine dummy(f)
real, dimension(:) :: f          ! assumed shape array
```

One can query the size of an array as follows:

```
integer :: i
i = size(f)
```

One can also declare temporary arrays inside procedures

```
subroutine dummy(f)
real, dimension(0:) :: f              ! lower bound of f is 0
real, dimension(size(f)) :: temp      ! automatic array
```

However, you have to be more careful when using Fortran90 arrays in procedures. Specifically, the procedures using such arrays **must** either have an explicit interface block whenever used or be in a module. This is because the compiler needs to know whether to pass the address of the array (as in Fortran77), or a descriptor with the hidden information (as in Fortran90). If you put all your procedures in modules, then you never have to worry.

Derived Types

Derived types are a named group of other types which can be created by a user to organize information.

```
type ordered_real
  real :: value
  integer :: key
end type
```

This type contains one real value and one integer. Such a type might be used to keep track of the order of a set of reals.

Derived types are similar to structs or records in other languages, and are a very important concept in modern programming.

To create a variable of this type:

```
type (ordered_real) :: a, b, c
a = ordered_real(1.,1)        ! structure constructor
```

One can also access components of this type as follows:

```
b%value = 2.; b%key = 5
```

And copy all the elements

```
c = a                    ! assignment operator (=) is defined
                         ! other operators (+,*) not defined
```

Why are derived types so important? Because they allow us to treat variables more abstractly.

Suppose, for example, we have a Fortran77 graphics subroutine with the following interface:

```
subroutine DISPR(f,label,scale,clip,marker,nx,nxv,ngs)
character(len=80) :: label
integer :: scale, clip, marker, nx, nxv, ngs
real, dimension(nxv,ngs) :: f
end subroutine
```

Four of these arguments are used to describe plots. Since they will always be used together, we can define a type so we can always refer to them as a unit:

```
type graf1d
   character(len=80) :: label
   integer :: scale, clip, marker
end type graf1d
```

Then instead of calling the function DISPR, we can create a new function dispr_f90:

```
 subroutine dispr_f90(f,nx,graf_params)
 type (graf1d) :: graf_params
 real, dimension(:,:) :: f    ! nxv, ngs not needed anymore
 call DISPR(f,grparams%label,grparams%scale,
&grparams%clip,grparams%marker,nx,size(f,1),size(f,2))
```

which is much easier to use. We can also change the graf1d type without changing how dispr_f90 is called.

Introduction to Object-Oriented Concepts in Fortran95

Object-Oriented Programming (OOP) is a design philosophy for writing complex software. Its main tool is the **abstract data type**, which allows one to program in terms of higher level concepts than just numbers and arrays of numbers. In their mathematics, physicists are quite familiar with the power of abstraction, e.g., we express physics equations using the curl operator, rather than writing out all the components. But we have not used such abstractions very much in our programming.

OOP includes a number of concepts which have proved useful in programming large projects. These are:

1. Information Hiding and Data Encapsulation

2. Function Overloading or Static Polymorphism

3. Abstract Datatypes, Classes and Objects

4. Inheritance

5. Dynamic Dispatch or Run-Time Polymorphism

Information Hiding and Data Encapsulation

Perhaps the most important concept is that of **information hiding**. This means that information which is required in only one procedure should not be made known to other procedures which do not need this information. Like the CIA, procedures should be informed of data only on a “need to know” basis. This philosophy simplifies programming, because there is less detail one must be concerned about in programming and less opportunities to make mistakes.

One way to achieve this is to **encapsulate** the data inside a derived type, and then allow only certain procedures (sometime called **methods**) to modify the data. One is prevented from modifying the data by any other means not provided by the programmer.

Such encapsulation permits **separation of concerns**. One can separately write and debug pieces of a large program, without worrying about a new procedure causing inadvertent damage to an older procedure. Writing complex program becomes an order N problem, rather than an order N^2 problem.

Let's look at an example of what this means. Consider the following **interface** to a legacy Fortran77 fft procedure:

```
subroutine fft1r(f,t,isign,mixup,sct,indx,nx,nxh)
integer isign, indx, nx, nxh, mixup(nxh)
real f(nx)
complex sct(nxh), t(nxh)
 ... rest of procedure goes here
```

In this procedure, f is the input (and output) data, t is a temporary work array, mixup is a bit reversed table, sct is a sine/cosine table, indx is the power of 2 defining the length of the transpose, nx is the size of the f, and nxh is size of the remaining data, and isign is either the direction of the transform (-1,1) or a request to initialize the tables (0).

To use this fft, one must get all of this data correct, there are many opportunities for mistakes. However, most of this data is relevant only internal details of performing the fft. The programmer only wants to worry about the data f and the direction of the transpose. Life would be much simpler if one could merely call

```
call fft1(f,isign)
```

without having to worry about the other details.

One of the reason all these details are exposed is that Fortran77 did not allow dynamic arrays. By using automatic and allocatable arrays, one can easily hide the scratch array t and the tables mixup and sct inside a wrapper function:

```
subroutine fft1(f,indx,isign)
integer :: indx, isign, nx, nxh
real, dimension(:) ::  f
complex, dimension(size(f)/2) :: t
integer, dimension(:), allocatable, save :: mixup
complex, dimension(:), allocatable, save :: sct
nx = size(f); nxh = nx/2
if (isign==0) allocate(mixup(nxh),sct(nxh))
call fft1r(f,t,isign,mixup,sct,indx,nx,nxh)
```

Thus the programmer does not have to worry about these things anymore and there is less opportunity for error.

We have successfully hidden from the programmer details about the fft that are not necessary to know to use the fft. Now the interface is much simpler and less error prone:

```
call fft1(f,indx,isign)
```

If one gets the interface down to its bare essentials, then it is unlikely to change in the future, even if the internal details of the procedure do change. For example, suppose on a given computer, there was an optimized fft which was much faster than the legacy fft1r. One could now replace the call to fft1r inside the wrapper function, and the users of the wrapper function would not have to change anything in their code.

```
subroutine fft1(f,indx,isign)
 ...
call faster_fft1r(f,......)       ! different internal arguments
end subroutine

call fft1(f,indx,isign)      ! Note the call does not change
```

Thus encapsulation allows one to change the implementation details of a procedure without impacting the rest of the program. This also allows concurrent development: different programmers can be modifying different pieces of a large program, without worrying about getting in each other's way, so long as the interfaces do not change.

Abstract Datatypes, Classes and Objects

An **abstract data type** or **class** encapsulates a user defined data type along with the operations that one can perform on that type.

The main value of classes is to encapsulate and hide the complex details of a set of related operations while presenting a simplified set of functions for the programmer to use. This encapsulation prevents inadvertent modification of internal data and also allows the implementation details to be changed without impacting the usage of the class.

As a result, it is much easier for different programmers to implement different classes without getting in each others way, and it (usually) means that once a class is debugged, one does not have to worry about it further when debugging other new code. As a result, more complex computational projects can be attempted.

Here is a basic electrostatic particle class

It contains a type:

```
    module species2d_class
     type species2d
        real :: qm, qbm, dt
        integer :: ipbc, nbmax, np, npp
        integer :: popt, dopt, sortime
        real, dimension(:,:,:), pointer :: part
     end type species2d
```

qm = charge on particle, in units of e
qbm = particle charge/mass ratio
dt = time interval between successive calculations
ipbc = particle boundary flag
nbmax = buffer size for passing particles between procs
np = total number of particles
npp = number of particles in this processor
popt = particle optimization flag
dopt = charge deposit optimization flag
sortime = number of time steps between particle sorting
part(:,:,m) = particle coordinates in this processor

And it contains functions which operate on that type:

Constructors and destructors:

```
  subroutine new_species2d(this,qm,qbm,ipbc,popt,dopt,
sortime)

  subroutine del_species2d(this)
```

Initialization functions:

```
   subroutine init_species(this,nspace,fdist)
```

General functions:

```
   subroutine qdeposit(this,qfield)
! deposits charge on a grid

   subroutine push(this,efield,ek);
! updates the particle co-ordinates

   subroutine pmove(this,nspace)
! moves particles into appropriate processor

   subroutine sortp(this,nspace)
! sorts particles by y grid for cache optimization

   subroutine wrdata(this,iunit);
! this subroutine collects distributed particle data part
! and writes it to a file

   subroutine rddata(this,iunit,ierror)
! this subroutine reads particle data from a file and
distributes it
```

A variable of this type is called an object. It is declared as follows:

```
type (species2d) :: electrons
```

The components of a derived type are called the **class data members**. The procedures defined in the class are called **class member functions**. *Generally, they provide the only means by which one can manipulate species2d objects*.

Typical usage:

```
 type (species2d) :: electrons
call new_species2d(electrons,qme,qbme,ipbc,sortime,
                         popt,dopt,sortime)
call init_species(electrons,nspace,fdist)
call push(electrons,efield,electron_ke)
call qdeposit(electrons,qfield)
```

The public interface to a class presents an abstract type to the outside world. By requiring the outside world to use only these interfaces, keeping the internal details of a class private, the internal data cannot be corrupted, and the implementation of methods can be changed without impacting others.

Inheritance

Inheritance is a mechanism to create a hierarchy of classes in which a **parent** (or base) class contains the common properties of the hierarchy and **child** (or derived) classes can modify (or specialize) these properties. The value of inheritance is to avoid duplicating code when creating classes which are similar to one another.

In object-oriented languages, inheritance specifically refers to a data relationship where the child class **contains** the data and functions of the parent. This allows the child to act as a parent for those functions that the child is not modifying. In other words, an inheritance relationship is like that of a Russian matrioska doll, where one doll fits inside another.

Inheritance is automated in object-oriented languages because of this special relationship. In Fortran95, however, inheritance must be manually constructed.

To see how this works, let us continue with the particle example. Now suppose we want to create a special class for relativistic, electromagnetic particles which contains additional information specific to pushing particles with electromagnetic fields, namely the speed of light and a flag for choosing a current deposit method, but the other information for a particle is the same as before, and we would like to avoid rewriting that code for the electromagnetic class.

There are several ways to achieve this. One way is to define an rel_em_species2d class to explicitly contain a speciesd2d object in its class definition:

```
module rel_em_species2d_class
 use species2d_class
 type rel_em_species2d
     type (species2d) :: es
    real :: ci
    integer :: djopt
 end type rel_em_species2d
```

ci = reciprical of velocity of light
djopt = current deposit optimization flag

We also have to define two new methods:

```
subroutine push(this,efield,bfield,electron_ke)
subroutine jdeposit(this,cfield)
```

We want all the other methods defined to work with electrostatic particles to also work with the new particles. In other words, the following functions must also work:

```
subroutine qdeposit(this,qfield)
subroutine pmove(this,nspace)
subroutine sortp(this,nspace)
subroutine wrdata(this,iunit);
subroutine rddata(this,iunit,ierror)
```

One way is to **delegate** the work to the parent class, by writing a one line subroutine for the new class, which merely calls the original subroutine for the old class:

```
subroutine qdeposit(this,qfield)
type (rel_em_species2d) :: this
call qdeposit(this%es,qfield)
end subroutine
```

In an OO language, these one line subroutines would not have to be written.

Typical usage:

```
type (rel_em_species2d) :: relectrons
call new_species2d(relectrons,qme,qbme,ipbc,sortime,
                   popt,dopt,sortime)
call init_species(relectrons,nspace,fdist)
call push(relectrons,efield,bfield,electron_ke)
call qdeposit(relectrons,qfield)
call jdeposit(relectrons,cfield)
```

Dynamic Dispatch or Run-Time Polymorphism

Run-time polymorphism (also known as **dynamic binding**) allows a single object name to refer to any member of an inheritance hierarchy and permits a procedure to resolve at run-time which actual object is being referred to. This is useful because it allows one to write programs in terms of a single type which would behave differently depending on the actual type. Object-oriented languages support this behavior. Fortran95 does not, with the exception of elemental functions.

In other words, we would like to be able to write a subroutine like this:

```
subroutine update(this,efield,bfield,ke)
type (any_kind_of_species) :: this
call push(this,efield,bfield,ke)      ! does the right stuff
```

which does something like this:

```
subroutine update(this,efield,bfield,ke)
type (any_kind_of_species) :: this
if (this==species2d) then
   call push(electrons,efield,ke)
else if (this==rel_em_species2d) then
   call push(electrons,efield,bfield,ke)
endif
end subroutine
```

There are techniques for accomplishing this in Fortran95, but I do not have time to go into them now.

UPIC Framework

A **framework** is a unified environment containing all the **components** needed for writing code for a specific problem domain. Its goal is the rapid construction of new codes by **reuse** of trusted modules.

UPIC Framework designed to help construct Plasma PIC calculations by student programmers.

Supports multiple numerical methods, different physics approximations, different numerical optimizations and implementations for different hardware.

Designed to hide the complexity of parallel processing.

The UPIC Framework has been used to build a number of new PIC codes:

1. QuickPIC, a quasi-static plasma accelerator code
2. QPIC, a quantum PIC code
3. BEPS, a computational lab for teaching plasma physics
4. HIPASS2, a space physics code
5. an unnamed cosmology code

For further information:

http://exodus.physics.ucla.edu/Fortran95/PSTIResearch
LecSeries1.html

Layered Approach (**software stack**)

Bottom Layer: optimized Fortran77 kernels.

• Well tested, often with years of use, don't need to fix it

Lower Layer: common utility functions

• Timing, trace facilities, tracking memory leaks

• Parallel I/O

Middle Layer: helper objects that work with Fortran90 arrays. They describe data, do not contain data

• Fortran90 arrays powerful, popular, well-understood

• Simpler environment for building new high level objects

• Hides parallel processing details

• Implements some polymorphism

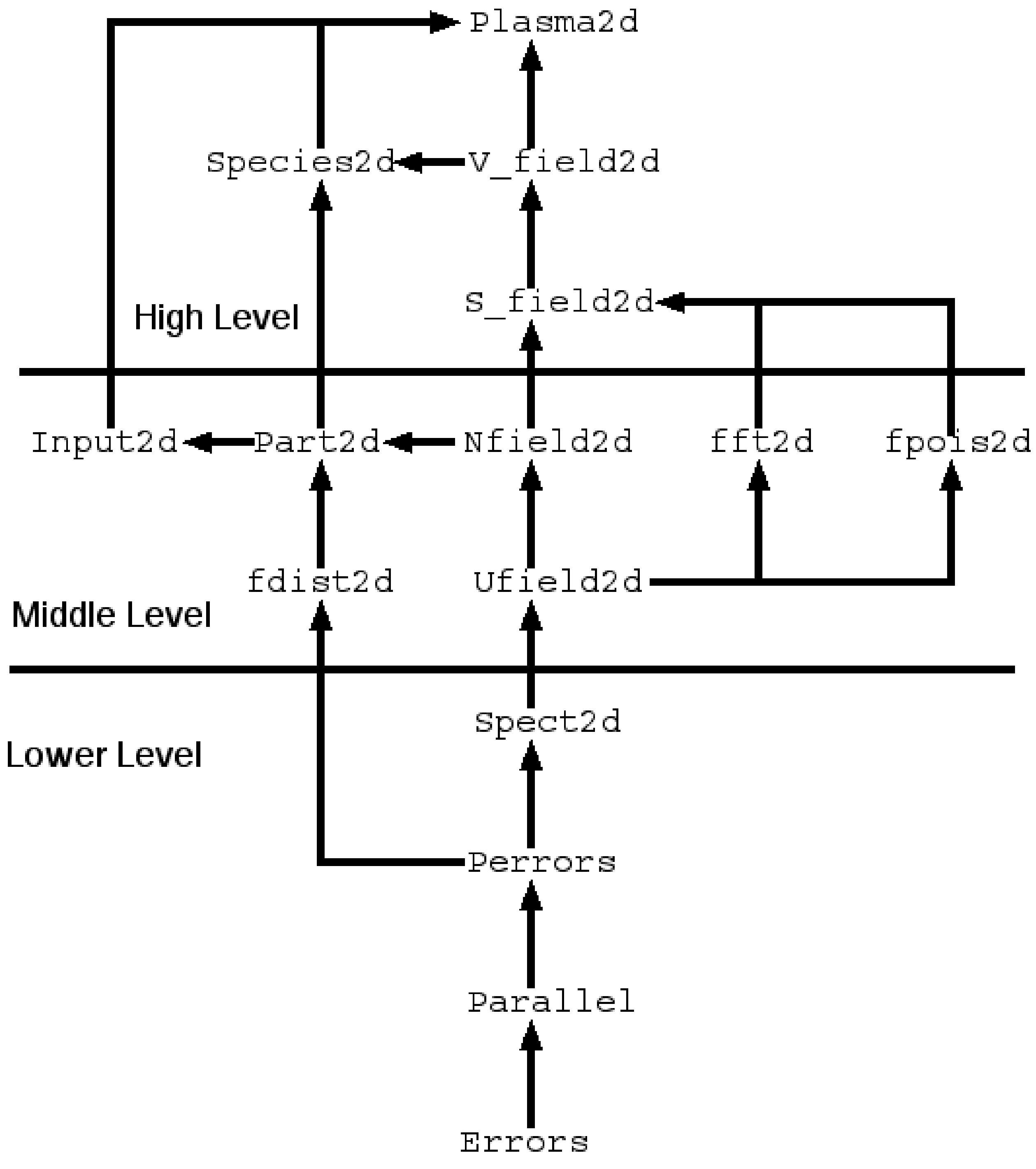
• Can be used in other codes and frameworks

Higher Layers: objects with properties

• Objects contain hidden pointers to all structures need to perform required operation

• Send message to object: "FFT yourself."

QuickPIC Framework Classes



Lower Level Classes in UPIC Framework

Errors class

This “class” provides support for debugging

Parallel class

This “class” provides basic support for parallel processing. Designed for spectral methods with the number of processors equal to a power of 2.

Perrors class

This “class” modifies the error class to provide support for parallel debugging.

Spect2d Class

This class encapsulates **basic information** about simulations which use 2d spectral methods for solutions.

Input2d class

This “class” defines the input namelist variables and provides default values for each of them.

Middle Level Classes in UPIC Framework

Ufield2d class

This class provides support for uniformly partitioned distributed scalar and vector arrays.

Nfield2d class

This class provides support for non-uniformly partitioned distributed scalar and vector arrays.

Fdist2d class

This middle level class provides functions to describe distributions of particles.

FFT2d class

This class provides functions to perform real to complex 2d FFTs for uniformly partitioned scalar or vector data.

Fpois2d class

This class provides functions to solve Poisson's equation in fourier space for scalar or vector data.

Part2d class

This middle level class contains information about particle properties and information needed to process particles.

High Level Classes in UPIC Framework

S_field2d class

This class provides support for scalar fields that have properties. Examples of such fields include charge density or potentials. This class encapsulates the information necessary for the fields to be able to perform various operations such as solving Poisson's equation.

V_field2d class

This class provides support for vector fields that have properties. Examples of such fields include current density or electric fields. This class is very similar to the scalar field class in functionality.

Species2d class

This class contains particle co-ordinates as well as a particle helper object which describes properties of a particle. There are three main methods in the class, depositing charge and current and pushing particles. The helper object determines what algorithms are used in each case.

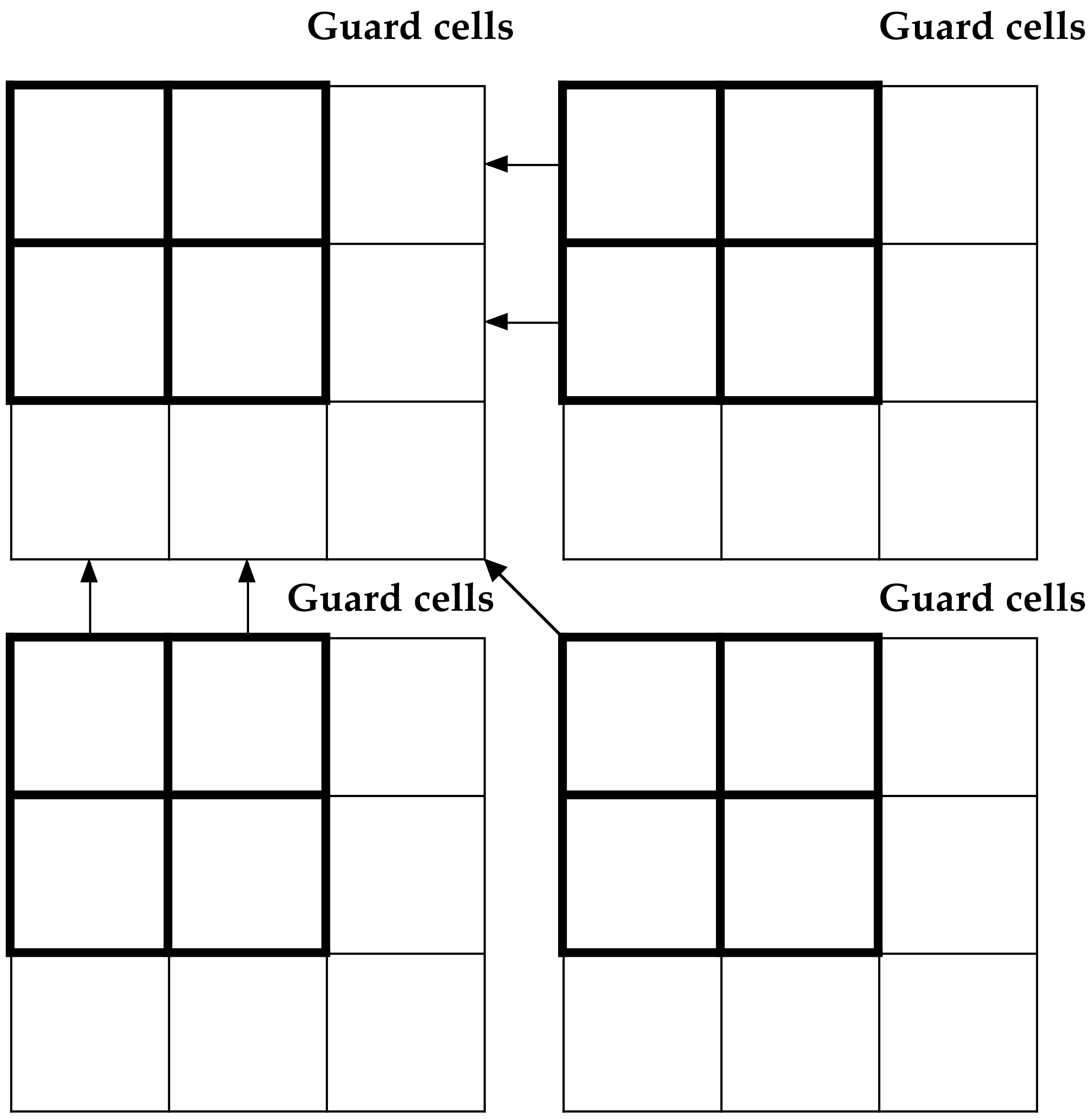
Plasma2d_class

This class encapsulates the entire code into one super class. One merely puts all the initial variables into a type.

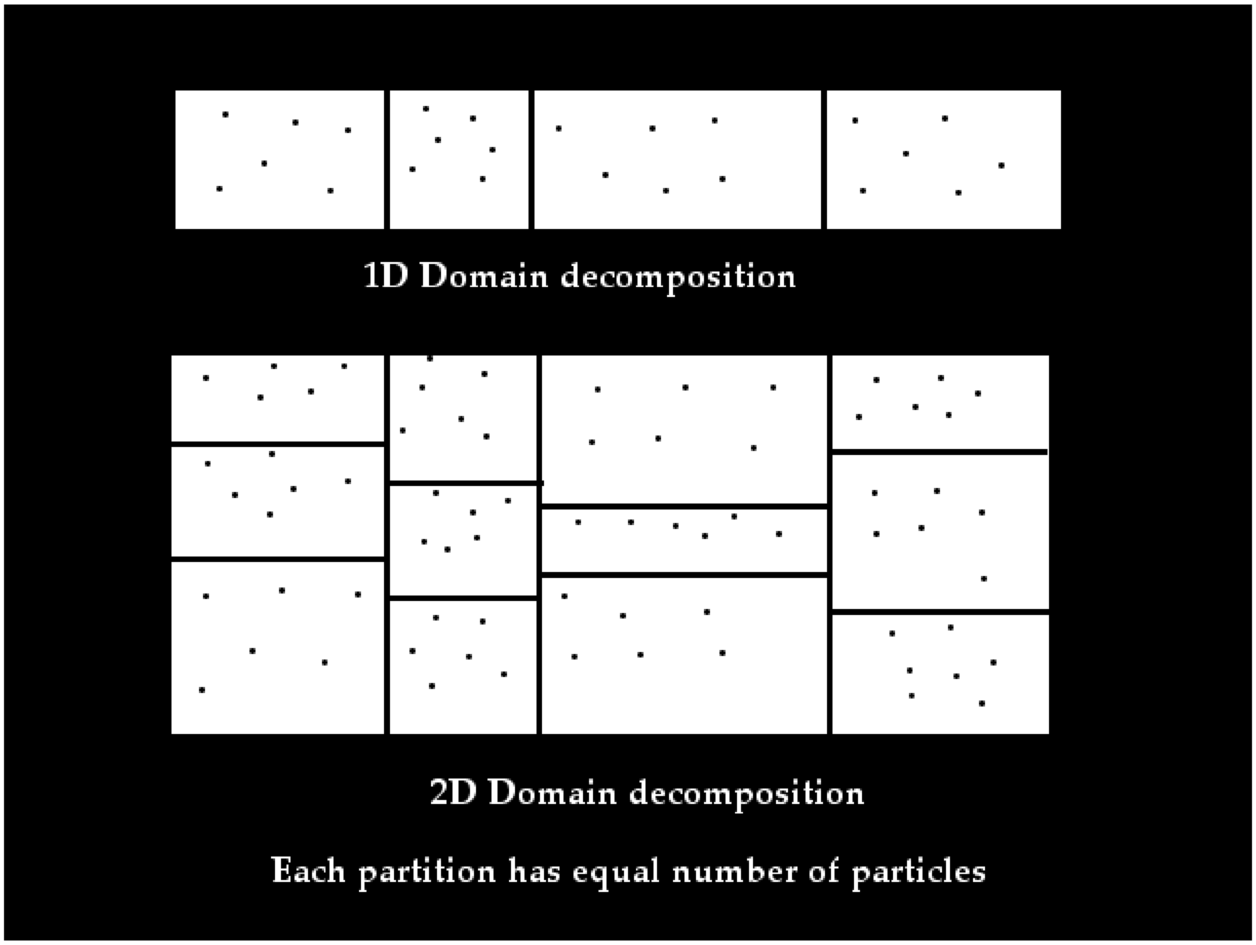
Here is the implementation of one of the functions:

```
         subroutine transp2d(source,f,dest,g)
! this subroutine performs transpose between different layouts
! source, dest = ufield2d descriptors of data
! f = source for transpose
! g = result of transpose
! kstrt = starting data block number, a global variable
        implicit none
         type (ufield2d), intent(in) :: source, dest
         complex, dimension(:,:,:) :: f, g
! local data
         integer :: nx, ny, nxv, nyv, kxp, kyp, kxpd, kypd
         integer :: jblok, kblok
         complex, dimension(dest%nd2p,source%nd2p,size(f,3)) :: s
         complex, dimension(dest%nd2p,source%nd2p,size(g,3)) :: t
         character(len=10), save :: sname = ':transp2d:'
        if (monitor==2) call werrfl(class//sname//' started')
! check for errors
         if (monitor > 0) then
         if ((source%partition /= 1) .or. (dest%partition /= 1)) then
           erstr = ' invalid/non-conforming partition'
            UFIELD2D_ERR = 6; EXCEPTION = EXCEPTION + 1
            call ehandler(EDEFAULT,class//sname//erstr); return
         endif
         if ((source%mshare /= 0) .or. (dest%mshare /= 0)) then
           erstr = ' non-conforming memory sharing'
            UFIELD2D_ERR = 7; EXCEPTION = EXCEPTION + 1
            call ehandler(EDEFAULT,class//sname//erstr); return
         endif
         endif
```

```
! obtain transpose arguments
         nxv = size(f,1); nyv = size(g,1)
         kxp = dest%nd2p; kyp = source%nd2p
         kxpd = size(g,2); kypd = size(f,2)
         jblok = dest%n2blok; kblok = source%n2blok
         nx = source%nd1; ny = source%nd2
         if (abs(source%layout - dest%layout)==1) then
! make sure arrays are conforming
            if ((source%nd1 /= dest%nd2) .or. (source%nd2 /= dest%nd1)) &
     &then
               erstr = ' non-conforming array'
               UFIELD2D_ERR = 8; EXCEPTION = EXCEPTION + 1
               call ehandler(EDEFAULT,class//sname//erstr); return
            endif
! perform transpose
           call PTPOSE(f,g,s,t,nx,ny,kstrt,nxv,nyv,kxp,kyp,kxpd,kypd,jb&
     &lok,kblok)
! unsupported transpose
         else
            erstr = ' unsupported transpose'
            UFIELD2D_ERR = 9; EXCEPTION = EXCEPTION + 1
            call ehandler(EDEFAULT,class//sname//erstr); return
         endif
        if (monitor==2) call werrfl(class//sname//' complete')
         end subroutine transp2d
```



Ufield2d Uniformly Partitioned Data



Nfield2d Non-uniformly Partitioned Data

```
module plasma2d_class        ! Super class
```

class library for describing high level 2d plasma objects

```
type plasma2d
  integer :: idproc, id0, nvp, mshare, nloop, itime
   integer :: ou, ov, or, os
   type (spect2d) :: space
   type (s_field2d) :: q, qi, den, pot
   type (v_field2d) :: fxy, bxy, cu, exyz, bxyz
   type (fdist2d) :: ebackg, ebeam, elects
   type (part2d) :: epart
   type (species2d) :: electrons
  real, dimension(:,:), pointer :: wt
  real, dimension(:,:,:), pointer :: fv
end type plasma2d
```

idproc = processor id in lgrp communicator
id0 = processor id in MPI_COMM_WORLD
nvp = number of real or virtual processors
mshare = (0,1) = (no,yes) architecture is shared memory
nloop = number of time steps in simulation
itime = current time step
ou, ov, or, os = output fortran unit numbers
space= helper objects for 2d spectral fields
q, qi, den, pot = scalar field objects
fxy, bxy, cu, exyz, bxyz = vector field objects
ebackg, ebeam, elects = objects for distribution functions
epart = helper object for electrons
electrons = electron object
wt = time history array for energies
fv = velocity distribution

This high level object contains all the variables describing a plasma, including high level fields and species.

There is one main method in the class, advancing the plasma state one time step.

Constructors:

```
      subroutine new_plasma2d(this)
! this subroutine creates a high level 2d plasma object
```

Destructor:

```
      subroutine del_plasma2d(this)
! delete a high level 2d plasma object
```

General functions:

```
      function update_plasma2d(this) result(done)
! this subroutine updates a 2d plasma object one time step
! done = current time step itime > 0, if no error
! otherwise = (-1,-2), indicating simulation is
(complete,aborted)

      subroutine printout_plasma2d(this)
! this subroutine prints out the data members of a
plasma2d object
```

Fortran90 Main Code:

```
      program simulation2d
!
      use plasma2d_class
      implicit none
      integer :: done = 0
      type (plasma2d) :: plasma
!
      call new_plasma2d(plasma)
      do while (done >= 0)
         done = update_plasma(plasma)
! one could process additonal events here
      enddo
      call del_plasma2d(plasma)
!
      end program simulation2d
```

C Main program:

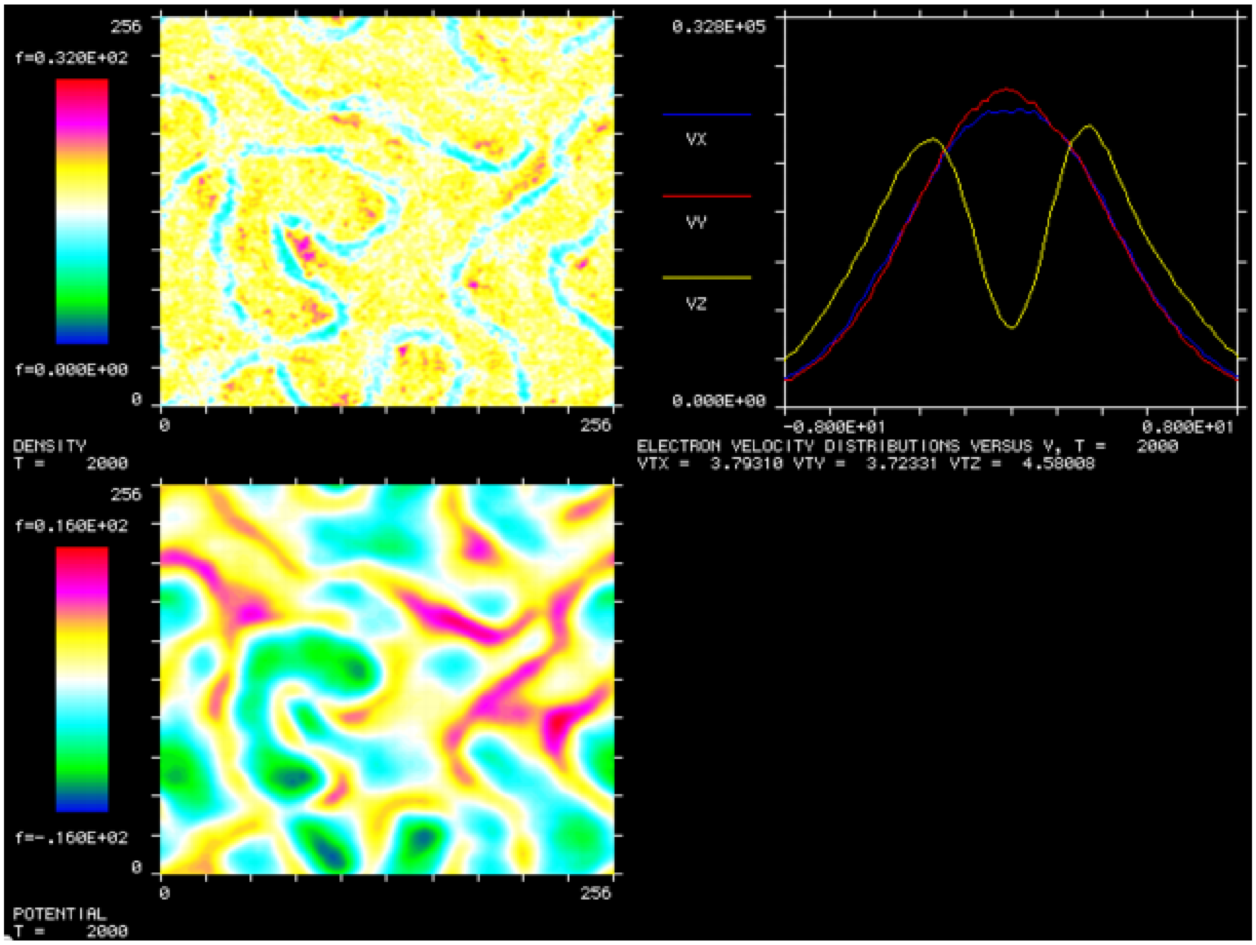
```
#include <stdlib.h>
#include <stdio.h>

/* prototypes for internal procedures */

void NEW_2DPLASMA(int *plasma_id);
void DEL_2DPLASMA(int *plasma_id);
int UPDATE_2DPLASMA(int *plasma_id);

int main(int argc, char *argv[])
{
   int done = 0, plasma;

   NEW_2DPLASMA(&plasma);
   while (done > 0) {
      done = UPDATE_2DPLASMA(&plasma);
/* one could process additonal events here */
   }
   DEL_2DPLASMA(&plasma);
   return 0;
}
```



Relativistic EM Two Stream Instability 2D BEPS Code